// robots.cpp

// Portions you are to complete are marked with a TODO: comment.

// We've provided some incorrect return statements (so indicated) just

// to allow this skeleton program to compile and run, albeit incorrectly.

// The first thing you probably want to do is implement the trivial

// functions (marked TRIVIAL). Then get Arena::display going. That gives

// you more flexibility in the order you tackle the rest of the functionality.

// As you finish implementing each TODO: item, remove its TODO: comment.

#include <iostream>

#include <string>

#include <random>

#include <utility>

#include <cstdlib>

using namespace std;

///////////////////////////////////////////////////////////////////////////

// Manifest constants

///////////////////////////////////////////////////////////////////////////

const int MAXROWS = 20; // max number of rows in the arena

const int MAXCOLS = 30; // max number of columns in the arena

const int MAXROBOTS = 100; // max number of robots allowed

const int UP = 0;

const int DOWN = 1;

const int LEFT = 2;

const int RIGHT = 3;

///////////////////////////////////////////////////////////////////////////

// Auxiliary function declarations

///////////////////////////////////////////////////////////////////////////

int decodeDirection(char dir);

int randInt(int min, int max);

void clearScreen();

///////////////////////////////////////////////////////////////////////////

// Type definitions

///////////////////////////////////////////////////////////////////////////

class Arena; // This is needed to let the compiler know that Arena is a

// type name, since it's mentioned in the Robot declaration.

class Robot

{

public:

// Constructor

Robot(Arena\* ap, int r, int c);

// Accessors

int row() const;

int col() const;

// Mutators

void move();

bool getAttacked(int dir);

private:

Arena\* m\_arena;

int m\_row;

int m\_col;

bool damaged;

// done.

// TODO: You'll probably find that a robot object needs an additional

// data member to support your implementation of the behavior affected

// by being hit and taking damage.

};

class Player

{

public:

// Constructor

Player(Arena \*ap, int r, int c);

// Accessors

int row() const;

int col() const;

int age() const;

bool isDead() const;

// Mutators

void stand();

void moveOrAttack(int dir);

void setDead();

private:

Arena\* m\_arena;

int m\_row;

int m\_col;

int m\_age;

bool m\_dead;

};

class Arena

{

public:

// Constructor/destructor

Arena(int nRows, int nCols);

~Arena();

// Accessors

int rows() const;

int cols() const;

Player\* player() const;

int robotCount() const;

int nRobotsAt(int r, int c) const;

bool determineNewPosition(int& r, int& c, int dir) const;

void display() const;

// Mutators

bool addRobot(int r, int c);

bool addPlayer(int r, int c);

bool attackRobotAt(int r, int c, int dir);

bool moveRobots();

private:

int m\_rows;

int m\_cols;

Player\* m\_player;

Robot\* m\_robots[MAXROBOTS];

int m\_nRobots;

};

class Game

{

public:

// Constructor/destructor

Game(int rows, int cols, int nRobots);

~Game();

// Mutators

void play();

private:

Arena\* m\_arena;

};

///////////////////////////////////////////////////////////////////////////

// Robot implementation

///////////////////////////////////////////////////////////////////////////

Robot::Robot(Arena\* ap, int r, int c)

{

if (ap == nullptr)

{

cout << "\*\*\*\*\* A robot must be in some Arena!" << endl;

exit(1);

}

if (r < 1 || r > ap->rows() || c < 1 || c > ap->cols())

{

cout << "\*\*\*\*\* Robot created with invalid coordinates (" << r << ","

<< c << ")!" << endl;

exit(1);

}

m\_arena = ap;

m\_row = r;

m\_col = c;

}

int Robot::row() const

{

return m\_row;

}

int Robot::col() const

{

return this->m\_col;

}

void Robot::move()

{

// Attempt to move in a random direction; if we can't move, don't move

int dir = randInt(0, 3); // dir is now UP, DOWN, LEFT, or RIGHT

if (dir == LEFT && (m\_col - 1) != m\_arena->cols())

m\_col--;

else if (dir == RIGHT && (m\_col + 1) != m\_arena->cols())

m\_col++;

else if (dir == UP && (m\_row - 1) != m\_arena->rows())

m\_row--;

else if (dir == DOWN && (m\_row + 1) != m\_arena->rows())

m\_row++;

// done-ish

// TODO: Attempt to move in direction dir; if we can't move, don't move.

}

bool Robot::getAttacked(int dir) // return true if dies

{

if (damaged)

return true;

else

{

if (m\_arena->determineNewPosition(m\_row, m\_col, dir))

{

damaged = true;

return false;

}

else

return true;

}

// done-ish

// TODO: If the robot has been attacked once before, return true

// (since a second attack kills a robot). Otherwise, if possible, move

// the robot in one position in direction dir and return false (since

// it survived the damage). Otherwise, do not move, but return true

// (since the momentum from the blow would bump the robot against the

// wall, dealing it additional fatal damage).

}

///////////////////////////////////////////////////////////////////////////

// Player implementations

///////////////////////////////////////////////////////////////////////////

Player::Player(Arena\* ap, int r, int c)

{

if (ap == nullptr)

{

cout << "\*\*\*\*\* The player must be in some Arena!" << endl;

exit(1);

}

if (r < 1 || r > ap->rows() || c < 1 || c > ap->cols())

{

cout << "\*\*\*\* Player created with invalid coordinates (" << r

<< "," << c << ")!" << endl;

exit(1);

}

m\_arena = ap;

m\_row = r;

m\_col = c;

m\_age = 0;

m\_dead = false;

}

int Player::row() const

{

return m\_row;

// done.

// TODO: TRIVIAL: Return what row the player is at.

}

int Player::col() const

{

return m\_col;

// done.

// TODO: TRIVIAL: Return what column the player is at.

}

int Player::age() const

{

return m\_age;

// done.

// TODO: TRIVIAL: Return the player's age.

}

void Player::stand()

{

m\_age++;

}

void Player::moveOrAttack(int dir)

{

m\_age++;

if (m\_arena->determineNewPosition(m\_row, m\_col, dir))

{

if (m\_arena->nRobotsAt(m\_row, m\_col) > 0)

{

m\_arena->attackRobotAt(m\_row, m\_col, dir);

switch (dir) // undo the move that determineNewPosition set, because you attacked instead of moved

{

case UP: m\_row++; break;

case DOWN: m\_row--; break;

case LEFT: m\_col++; break;

case RIGHT: m\_col--; break;

}

}

}

//done.

// TODO: If there is a robot adjacent to the player in the direction

// dir, attack it. Otherwise, move the player to that position if

// possible (i.e., if the move would not be off the edge of the arena).

}

bool Player::isDead() const

{

return m\_dead;

}

void Player::setDead()

{

m\_dead = true;

}

///////////////////////////////////////////////////////////////////////////

// Arena implementations

///////////////////////////////////////////////////////////////////////////

Arena::Arena(int nRows, int nCols)

{

if (nRows <= 0 || nCols <= 0 || nRows > MAXROWS || nCols > MAXCOLS)

{

cout << "\*\*\*\*\* Arena created with invalid size " << nRows << " by "

<< nCols << "!" << endl;

exit(1);

}

m\_rows = nRows;

m\_cols = nCols;

m\_player = nullptr;

m\_nRobots = 0;

}

Arena::~Arena()

{

delete m\_player;

for (int k = 0; k < m\_nRobots; k++)

{

delete m\_robots[k];

}

// done.

// TODO: Delete the player and all remaining dynamically allocated robots.

}

int Arena::rows() const

{

return m\_rows;

// done.

// TODO: TRIVIAL: Return the number of rows in the arena.

}

int Arena::cols() const

{

return m\_cols;

}

Player\* Arena::player() const

{

return m\_player;

}

int Arena::robotCount() const

{

return m\_nRobots;

}

int Arena::nRobotsAt(int r, int c) const

{

int counter = 0;

for (int k = 0; k < m\_nRobots; k++)

{

if (m\_robots[k]->col() == c && m\_robots[k]->row() == r)

counter++;

}

return counter;

// done.

// TODO: Return the number of robots at row r, column c.

}

bool Arena::determineNewPosition(int& r, int& c, int dir) const

{

//done.

// TODO: If a move from row r, column c, one step in direction dir

// would go off the edge of the arena, leave r and c unchanged and

// return false. Otherwise, set r or c so that row r, column c, is

// now the new position resulting from the proposed move, and

// return true.

switch (dir)

{

case UP:

if ((r - 1) < 1)

return false;

else r--;

break;

// done.

// TODO: Implement the behavior if dir is UP.

case DOWN:

if ((r + 1) >= m\_rows)

return false;

else r++;

break;

case LEFT:

if ((c - 1) < 1)

return false;

else c--;

break;

case RIGHT:

if ((c + 1) >= m\_cols)

return false;

else c++;

// done.

// TODO: Implement the other directions.

break;

default:

return false;

}

return true;

}

void Arena::display() const

{

// Position (row,col) in the arena coordinate system is represented in

// the array element grid[row-1][col-1]

char grid[MAXROWS][MAXCOLS];

int r, c;

// Fill the grid with dots

for (r = 0; r < rows(); r++)

for (c = 0; c < cols(); c++)

grid[r][c] = '.';

// Indicate each robot's position

for (int k = 0; k < m\_nRobots; k++)

{

int nRobots = nRobotsAt(m\_robots[k]->row(), m\_robots[k]->col());

switch (nRobots)

{

case 1:

grid[(m\_robots[k]->row()) - 1][(m\_robots[k]->col()) - 1] = 'R';

break;

case 2:

grid[(m\_robots[k]->row()) - 1][(m\_robots[k]->col()) - 1] = '2';

break;

case 3:

grid[(m\_robots[k]->row()) - 1][(m\_robots[k]->col()) - 1] = '3';

break;

case 4:

grid[(m\_robots[k]->row()) - 1][(m\_robots[k]->col()) - 1] = '4';

break;

case 5:

grid[(m\_robots[k]->row()) - 1][(m\_robots[k]->col()) - 1] = '5';

break;

case 6:

grid[(m\_robots[k]->row()) - 1][(m\_robots[k]->col()) - 1] = '6';

break;

case 7:

grid[(m\_robots[k]->row()) - 1][(m\_robots[k]->col()) - 1] = '7';

break;

case 8:

grid[(m\_robots[k]->row()) - 1][(m\_robots[k]->col()) - 1] = '8';

break;

default:

grid[(m\_robots[k]->row()) - 1][(m\_robots[k]->col()) - 1] = '9';

break;

}

}

//done.

// TODO: If one robot is at some grid point, set the char to 'R'.

// If it's 2 through 8, set it to '2' through '8'.

// For 9 or more, set it to '9'.

// Indicate player's position

if (m\_player != nullptr)

{

// Set the char to '@', unless there's also a robot there,

// in which case set it to '\*'.

char& gridChar = grid[m\_player->row() - 1][m\_player->col() - 1];

if (gridChar == '.')

gridChar = '@';

else

gridChar = '\*';

}

// Draw the grid

clearScreen();

for (r = 0; r < rows(); r++)

{

for (c = 0; c < cols(); c++)

cout << grid[r][c];

cout << endl;

}

cout << endl;

// Write message, robot, and player info

cout << endl;

cout << "There are " << robotCount() << " robots remaining." << endl;

if (m\_player == nullptr)

cout << "There is no player." << endl;

else

{

if (m\_player->age() > 0)

cout << "The player has lasted " << m\_player->age() << " steps." << endl;

if (m\_player->isDead())

cout << "The player is dead." << endl;

}

}

bool Arena::addRobot(int r, int c)

{

if (m\_nRobots == MAXROBOTS)

return false;

else

{

m\_robots[m\_nRobots] = new Robot(this, r, c);

m\_nRobots++;

return true;

}

//done.

// TODO: If MAXROBOTS have already been added, return false. Otherwise,

// dynamically allocate a new robot at coordinates (r,c). Save the

// pointer to the newly allocated robot and return true.

}

bool Arena::addPlayer(int r, int c)

{

// Don't add a player if one already exists

if (m\_player != nullptr)

return false;

// Dynamically allocate a new Player and add it to the arena

m\_player = new Player(this, r, c);

return true;

}

bool Arena::attackRobotAt(int r, int c, int dir)

{

for (int k = 0; k < m\_nRobots; k++)

{

if (this->nRobotsAt(r, c) > 0)

{

if (m\_robots[k]->row() == r && m\_robots[k]->col() == c)

{

bool damageFatal = m\_robots[k]->getAttacked(dir);

if (damageFatal)

{

delete m\_robots[k];

m\_robots[k] = m\_robots[m\_nRobots - 1];

m\_nRobots--;

return true;

}

}

else return false;

}

else return false;

}

}

// done-ish

// TODO: Attack one robot at row r, column c if at least one is at

// that position. If the robot does not survive the damage, destroy the

// robot object, removing it from the arena, and return true. Otherwise,

// return false (no robot at (r,c), or robot didn't die)

bool Arena::moveRobots()

{

for (int k = 0; k < m\_nRobots; k++)

{

m\_robots[k]->move();

if (m\_robots[k]->row() == m\_player->row() && m\_robots[k]->col() == m\_player->col())

m\_player->setDead();

//done.

// TODO: Have the k-th robot in the arena make one move.

// If that move results in that robot being in the same

// position as the player, the player dies.

}

// return true if the player is still alive, false otherwise

return !m\_player->isDead();

}

///////////////////////////////////////////////////////////////////////////

// Game implementations

///////////////////////////////////////////////////////////////////////////

Game::Game(int rows, int cols, int nRobots)

{

if (nRobots < 0)

{

cout << "\*\*\*\*\* Cannot create Game with negative number of robots!" << endl;

exit(1);

}

if (nRobots > MAXROBOTS)

{

cout << "\*\*\*\*\* Trying to create Game with " << nRobots

<< " robots; only " << MAXROBOTS << " are allowed!" << endl;

exit(1);

}

if (rows == 1 && cols == 1 && nRobots > 0)

{

cout << "\*\*\*\*\* Cannot create Game with nowhere to place the robots!" << endl;

exit(1);

}

// Create arena

m\_arena = new Arena(rows, cols);

// Add player

int rPlayer = randInt(1, rows);

int cPlayer = randInt(1, cols);

m\_arena->addPlayer(rPlayer, cPlayer);

// Populate with robots

while (nRobots > 0)

{

int r = randInt(1, rows);

int c = randInt(1, cols);

// Don't put a robot where the player is

if (r == rPlayer && c == cPlayer)

continue;

m\_arena->addRobot(r, c);

nRobots--;

}

}

Game::~Game()

{

delete m\_arena;

}

void Game::play()

{

Player\* p = m\_arena->player();

if (p == nullptr)

{

m\_arena->display();

return;

}

do

{

m\_arena->display();

cout << endl;

cout << "Move (u/d/l/r//q): ";

string action;

getline(cin, action);

if (action.size() == 0) // player stands

p->stand();

else

{

switch (action[0])

{

default: // if bad move, nobody moves

cout << '\a' << endl; // beep

continue;

case 'q':

return;

case 'u':

case 'd':

case 'l':

case 'r':

p->moveOrAttack(decodeDirection(action[0]));

break;

}

}

m\_arena->moveRobots();

} while (!m\_arena->player()->isDead() && m\_arena->robotCount() > 0);

m\_arena->display();

}

///////////////////////////////////////////////////////////////////////////

// Auxiliary function implementations

///////////////////////////////////////////////////////////////////////////

int decodeDirection(char dir)

{

switch (dir)

{

case 'u': return UP;

case 'd': return DOWN;

case 'l': return LEFT;

case 'r': return RIGHT;

}

return -1; // bad argument passed in!

}

// Return a random int from min to max, inclusive

int randInt(int min, int max)

{

if (max < min)

swap(max, min);

static random\_device rd;

static mt19937 generator(rd());

uniform\_int\_distribution<> distro(min, max);

return distro(generator);

}

///////////////////////////////////////////////////////////////////////////

// main()

///////////////////////////////////////////////////////////////////////////

void doBasicTests();

int main()

{

//doBasicTests(); // Remove this line after completing test.

//return 0; // Remove this line after completing test.

// Create a game

// Use this instead to create a mini-game: Game g(3, 4, 2);

Game g(7, 8, 25);

// Play the game

g.play();

}

///////////////////////////////////////////////////////////////////////////

// clearScreen implementation

///////////////////////////////////////////////////////////////////////////

// DO NOT MODIFY OR REMOVE ANY CODE BETWEEN HERE AND THE END OF THE FILE!!!

// THE CODE IS SUITABLE FOR VISUAL C++, XCODE, AND g++ UNDER LINUX.

// Note to Xcode users: clearScreen() will just write a newline instead

// of clearing the window if you launch your program from within Xcode.

// That's acceptable. (The Xcode output window doesn't have the capability

// of being cleared.)

#ifdef \_MSC\_VER // Microsoft Visual C++

#include <windows.h>

void clearScreen()

{

HANDLE hConsole = GetStdHandle(STD\_OUTPUT\_HANDLE);

CONSOLE\_SCREEN\_BUFFER\_INFO csbi;

GetConsoleScreenBufferInfo(hConsole, &csbi);

DWORD dwConSize = csbi.dwSize.X \* csbi.dwSize.Y;

COORD upperLeft = { 0, 0 };

DWORD dwCharsWritten;

FillConsoleOutputCharacter(hConsole, TCHAR(' '), dwConSize, upperLeft,

&dwCharsWritten);

SetConsoleCursorPosition(hConsole, upperLeft);

}

/\*

#include <type\_traits>

#include <cassert>

#define CHECKTYPE(c, f, r, a) \

static\_assert(std::is\_same<decltype(&c::f), r (c::\*)a>::value, \

"FAILED: You changed the type of " #c "::" #f); \

[[gnu::unused]] r (c::\* xxx##c##\_##f) a = &c::f

void thisFunctionWillNeverBeCalled()

{

// If the student deleted or changed the interfaces to the public

// functions, this won't compile. (This uses magic beyond the scope

// of CS 31.)

Robot(static\_cast<Arena\*>(0), 1, 1);

CHECKTYPE(Robot, row, int, () const);

CHECKTYPE(Robot, col, int, () const);

CHECKTYPE(Robot, move, void, ());

CHECKTYPE(Robot, getAttacked, bool, (int));

Player(static\_cast<Arena\*>(0), 1, 1);

CHECKTYPE(Player, row, int, () const);

CHECKTYPE(Player, col, int, () const);

CHECKTYPE(Player, age, int, () const);

CHECKTYPE(Player, isDead, bool, () const);

CHECKTYPE(Player, stand, void, ());

CHECKTYPE(Player, moveOrAttack, void, (int));

CHECKTYPE(Player, setDead, void, ());

Arena(1, 1);

CHECKTYPE(Arena, rows, int, () const);

CHECKTYPE(Arena, cols, int, () const);

CHECKTYPE(Arena, player, Player\*, () const);

CHECKTYPE(Arena, robotCount, int, () const);

CHECKTYPE(Arena, nRobotsAt, int, (int, int) const);

CHECKTYPE(Arena, determineNewPosition, bool, (int&, int&, int) const);

CHECKTYPE(Arena, display, void, () const);

CHECKTYPE(Arena, addRobot, bool, (int, int));

CHECKTYPE(Arena, addPlayer, bool, (int, int));

CHECKTYPE(Arena, attackRobotAt, bool, (int, int, int));

CHECKTYPE(Arena, moveRobots, bool, ());

Game(1, 1, 1);

CHECKTYPE(Game, play, void, ());

}

void doBasicTests()

{

{

Arena a(10, 20);

assert(a.addPlayer(2, 6));

Player\* pp = a.player();

assert(pp->row() == 2 && pp->col() == 6 && !pp->isDead());

pp->moveOrAttack(UP);

assert(pp->row() == 1 && pp->col() == 6 && !pp->isDead());

pp->moveOrAttack(UP);

// assert(pp->row() == 1 && pp->col() == 6 && !pp->isDead());

pp->setDead();

// assert(pp->row() == 1 && pp->col() == 6 && pp->isDead());

}

{

Arena a(2, 2);

assert(a.addPlayer(1, 1));

assert(a.addRobot(2, 2));

Player\* pp = a.player();

assert(a.moveRobots());

assert(!pp->isDead());

for (int k = 0; k < 1000 && !pp->isDead() && a.moveRobots(); k++)

;

// assert(pp->isDead());

}

{

Arena a(2, 6);

assert(a.addPlayer(2, 1));

assert(a.addRobot(2, 3));

Player\* pp = a.player();

pp->moveOrAttack(RIGHT);

assert(a.robotCount() == 1 && a.nRobotsAt(2, 3) == 1);

pp->moveOrAttack(RIGHT);

assert(a.robotCount() == 1 && a.nRobotsAt(2, 4) == 1);

pp->moveOrAttack(RIGHT);

assert(a.robotCount() == 1 && a.nRobotsAt(2, 4) == 1);

pp->moveOrAttack(RIGHT);

assert(a.robotCount() == 0 && a.nRobotsAt(2, 4) == 0 && a.nRobotsAt(2, 5) == 0);

a.addRobot(1, 3);

assert(a.robotCount() == 1 && a.nRobotsAt(1, 3) == 1);

pp->moveOrAttack(UP);

// assert(a.robotCount() == 0 && a.nRobotsAt(1, 3) == 0);

}

} \*/

#else // not Microsoft Visual C++, so assume UNIX interface

#include <iostream>

#include <cstring>

#include <cstdlib>

void clearScreen() // will just write a newline in an Xcode output window

{

static const char\* term = getenv("TERM");

if (term == nullptr || strcmp(term, "dumb") == 0)

cout << endl;

else

{

static const char\* ESC\_SEQ = "\x1B["; // ANSI Terminal esc seq: ESC [

cout << ESC\_SEQ << "2J" << ESC\_SEQ << "H" << flush;

}

}

#endif